Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №7

Технології паралельних обчислень

Тема: Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірила:
Стеценко І.В.

Київ 2024

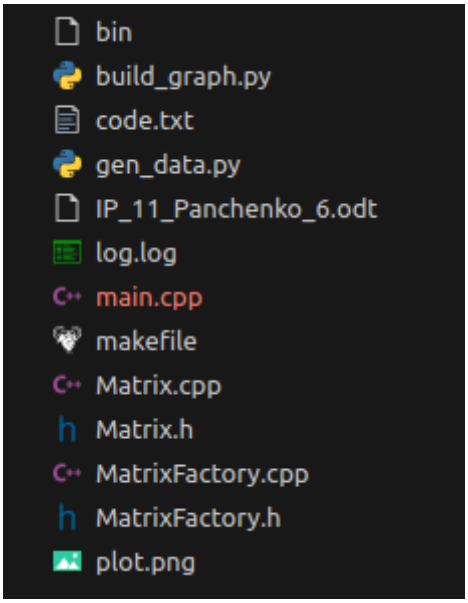
ЗМІСТ

1 ЗАВДАННЯ

1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. 40 балів.
3. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні методів обміну повідомленнями «один-до одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох». 60 балів.

2 ВИКОНАННЯ

2.1 Структура проєкту



Bin — бінарний файл готової програми.

Makefile — файл для перетворення .h, .cpp — файлів в об’єктні файли, для їхнього лінкування із зовнішніми бібліотеками в один бінарний файл.

Matrix.h та Matrix.cpp — клас матриці, що використовує бібліотеку boost/serialization для серіалізації.

MatrixFactory.h та MatrixFactory.cpp — клас-фабрика, для створення матриць.

Main.cpp — ресурсний файл, що власне відповідає за роботу зі стандартом MPI, використовуючи реалізацію boost/mpi.

Log.log — файл, куди записуються результати прогону бінарного файлу з переданими при його запуску параметрами.

gen_data.py — скрипт на мові Python для повторного запуску бінарного файлу.

build_graph.py — скрипт на мові Python для побудови графіку на основі даних, записаних до log.log.

2.2 Результати

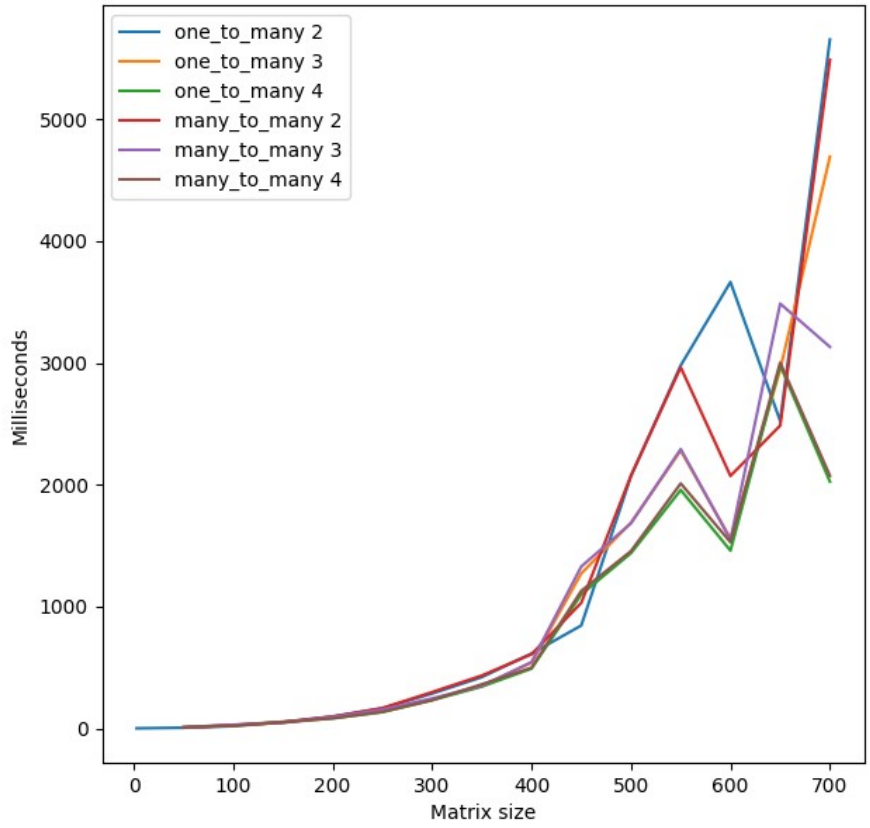


Рисунок 2.2.1 - Графік один-до-багатьох, багато-до-багатьох

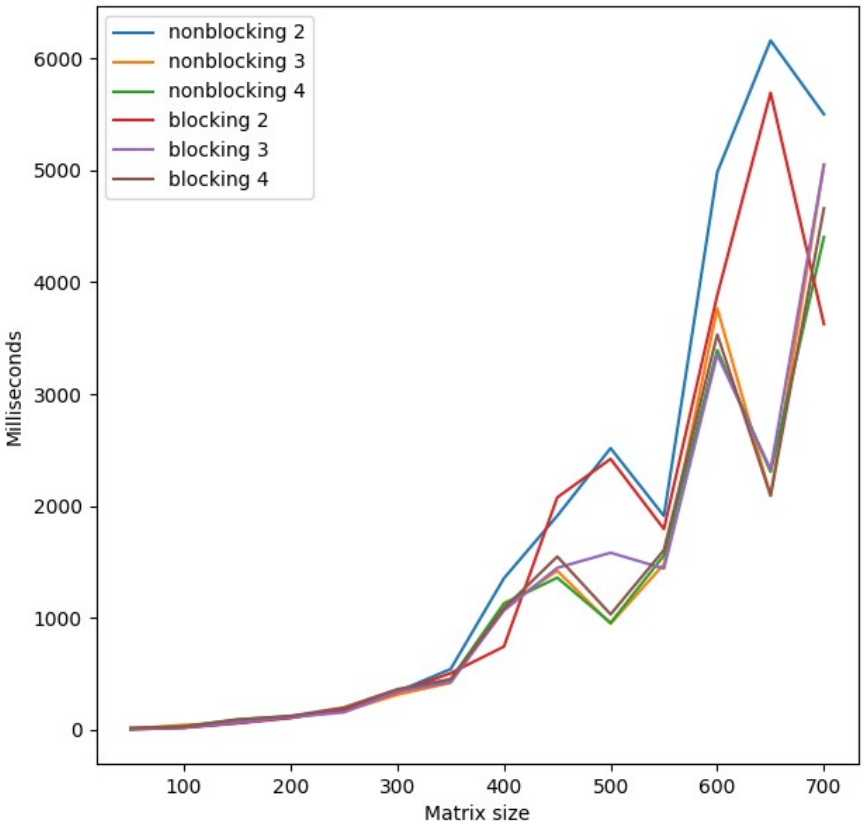


Рисунок 2.2.2 - Графік один-до-одного

У результаті отримали такий графіки, бачимо, що примітна різниця між двома підходами. Алгоритми один-до-одного суттєво більш викривлені та займають більше часу при тих самих розмірах та кількості потоків.

3 ВИСНОВОК

Під час лабораторної роботи опрацювали розробку паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності.

Побачили, що алгоритми один-до-багатьох, багато-до-багатьох швидше виконують множення матриць ніж один-до-одного.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-113 курсу

Панченка С. В

```
// ./Lab7/main.cpp

#define BOOST_LOG_DYN_LINK 1

#include <iostream>
#include <chrono>

#include <boost/mpi.hpp>

#include <boost/log/core.hpp>
#include <boost/log/trivial.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/sinks/text_file_backend.hpp>
#include <boost/log/utility/setup/file.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/record_ostream.hpp>

#include "MatrixFactory.h"
#include "Matrix.h"

namespace mpi = boost::mpi;
namespace logging = boost::log;

static constexpr auto MIN_VAL = 0;
static constexpr auto MAX_VAL = 1;

static const std::string NUMBER_OF_TASKS_LESS_EQUAL_ZERO = "Number of
tasks is less or equal zero";
static const std::string INVALID_ALG_TYPE = "incorrect algorithm type";
static const std::string MAT_SIZE_NEGATIVE = "Matrix size can not be negative";
```

```
static const std::string MILLIS = "ms";

static const std::string LOG_FILE = "log.log";
static const std::string LOG_FORMAT = "%Message%";
static constexpr auto LOG_OPEN_MODE = std::ios_base::app;

static const std::string ONE_TO_MANY = "one_to_many";
static const std::string MANY_TO_MANY = "many_to_many";

static constexpr char TAB = '\t';

void initLog();
void doMul(const mpi::communicator& world, unsigned matSize, const std::string&
algType);

int main(int argc, char* argv[]) {

	auto env = mpi::environment();
	auto world = mpi::communicator();

	if( world.size() - 1 <= 0) {
		throw
std::invalid_argument(NUMBER_OF_TASKS_LESS_EQUAL_ZERO);
	}

	initLog();

	auto matSize = static_cast<unsigned>(std::stoi(argv[1]));

	if(matSize < 0) {
		throw std::invalid_argument(MAT_SIZE_NEGATIVE);
	}
```

```
	auto algType = std::string(argv[2]);

	if(algType != ONE_TO_MANY && algType != MANY_TO_MANY) {
		throw std::invalid_argument(INVALID_ALG_TYPE);
	}

	doMul(world, matSize, algType);

	return 0;
}

void doMul(const mpi::communicator& world, unsigned matSize, const std::string&
algType) {
	auto rank = static_cast<unsigned>(world.rank());

	Matrix first, second;
	auto result = Matrix(matSize, matSize);

	if(rank == 0) {
		auto factory = MatrixFactory();
		first = factory.GenerateMatrix(matSize, matSize, MIN_VAL,
MAX_VAL);
		second = factory.GenerateMatrix(matSize, matSize, MIN_VAL,
MAX_VAL);
	}

	mpi::broadcast(world, second, 0);

	auto start = std::chrono::high_resolution_clock::now();

	auto nodesNum = static_cast<unsigned>(world.size());
	auto step = matSize / nodesNum;
```

```
auto extraSteps = matSize % nodesNum;

auto outProxy = Matrix();

if(rank == 0) {
	auto firstInnerMat = first.innerMat();
	auto proxis = std::vector<Matrix>(nodesNum);
	for(auto i = 0u; i < proxis.size(); ++i) {
		auto from = firstInnerMat.begin() + i * step;
		auto to = from + step;
		std::copy(from, to, std::back_inserter(proxis[i].innerMat()));
	}
	mpi::scatter(world, proxis, outProxy, 0);
} else {
	mpi::scatter(world, outProxy, 0);
}

auto fromRow = rank * step;
for(unsigned i = fromRow, r = 0; (i < fromRow + step) && (r < step); ++i) {
	for(auto j = 0; j < matSize; ++j) {
		for(auto k = 0; k < matSize; ++k) {
			result[i][j] += outProxy[r][k] * second[k][j];
		}
	}
}

std::vector<Matrix> subResults;

if(rank == 0) {
	// receive subResults to main thread from worker thread
	if(algType == ONE_TO_MANY) {
		mpi::gather(world, result, subResults, 0);
```

```
        } else if(algType == MANY_TO_MANY) {
            mpi::all_gather(world, result, subResults);
        }
        auto& sub = subResults.front();
        for(auto i = 1u; i < subResults.size(); ++i) {
            sub.sum(subResults[i]);
        }

        for(auto i = sub.rows() - extraSteps; i < sub.rows(); ++i) {
            for(auto j = 0; j < matSize; ++j) {
                for(auto k = 0; k < matSize; ++k) {
                    sub[i][j] += first[i][k] * second[k][j];
                }
            }
        }

        // std::cout<<first<<std::endl;
        // std::cout<<second<<std::endl;
        // std::cout<<sub<<std::endl;

        auto end = std::chrono::high_resolution_clock::now();
        auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();

        BOOST_LOG_TRIVIAL(info) << algType << TAB << matSize
            << TAB << world.size() << TAB << millis;
    } else {
        // send subResult of worker thread to the main thread
        if(algType == ONE_TO_MANY) {
            mpi::gather(world, result, 0);
        } else if(algType == MANY_TO_MANY) {
            mpi::all_gather(world, result, subResults);
```

```
            }
        }
}

void initLog() {
        logging::add_file_log(
                LOG_FILE,
                logging::keywords::open_mode = LOG_OPEN_MODE,
                logging::keywords::format = LOG_FORMAT
        );
    logging::core::get()->set_filter(
        logging::trivial::severity >= logging::trivial::info
    );
}


// ./Lab7/MatrixFactory.h

#ifndef LAB6_MATRIXFACTORY_H
#define LAB6_MATRIXFACTORY_H

#include <vector>
#include <random>

class Matrix;

class MatrixFactory {
        public:
        MatrixFactory()=default;
        virtual ~MatrixFactory()=default;

        public:
```

```
        virtual Matrix GenerateMatrix(unsigned rows, unsigned cols, double minVal, double maxVal);

        protected:
        std::random_device rd;
        std::mt19937 rng = std::mt19937(rd());
        std::uniform_real_distribution<double> uni = std::uniform_real_distribution<double>(0, 1);
};


#endif //LAB6_MATRIXFACTORY_H


// ./Lab7/Matrix.cpp

#include "Matrix.h"
#include <iostream>

static constexpr char COMMA_SYMBOL = ',';
static constexpr char OPEN_BRACKETS_SYMBOL = '{';
static constexpr char CLOSING_BRACKETS_SYMBOL = '}';
static constexpr char SPACE_SYMBOL = ' ';
static const std::string MATRICES_NOT_THE_SAME_SIZE = "Matrices are not the same size";

Matrix::Matrix(unsigned int rows, unsigned int cols) {
        mat.resize(rows);
        for(auto& row : mat) {
                row.assign(cols, 0);
        }
}
```

```
Matrix::InnerRow& Matrix::operator[](unsigned index) {
	return mat[index];
}

const Matrix::InnerRow& Matrix::operator[](unsigned index) const {
	return mat[index];
}

Matrix::InnerMat& Matrix::innerMat() {
	return mat;
}

const Matrix::InnerMat& Matrix::innerMat() const {
	return mat;
}

unsigned Matrix::rows() const {
	return mat.size();
}

unsigned Matrix::cols() const {
	return mat.front().size();
}

std::ostream& operator<<(std::ostream& out, const Matrix& matrix) {
	out << OPEN_BRACKETS_SYMBOL << SPACE_SYMBOL;
	for(auto i = 0u; i < matrix.rows() - 1; ++i) {
		out << OPEN_BRACKETS_SYMBOL << SPACE_SYMBOL;
		for(auto j = 0u; j < matrix.cols() - 1; ++j) {
			out << matrix[i][j] << COMMA_SYMBOL << SPACE_SYMBOL;
		}
```

```
            out << matrix[i][matrix.cols() - 1] << SPACE_SYMBOL;
            out << CLOSING_BRACKETS_SYMBOL << COMMA_SYMBOL <<
SPACE_SYMBOL;
        }
        out << OPEN_BRACKETS_SYMBOL << SPACE_SYMBOL;
        for(auto j = 0u; j < matrix.cols() - 1; ++j) {
            out << matrix[ matrix.rows() - 1][j] << COMMA_SYMBOL <<
SPACE_SYMBOL;
        }
        out << matrix[ matrix.rows() - 1][matrix.cols() - 1] << SPACE_SYMBOL;
        out << CLOSING_BRACKETS_SYMBOL << SPACE_SYMBOL;
        out << CLOSING_BRACKETS_SYMBOL << SPACE_SYMBOL;
        return out;
}

void Matrix::sum(const Matrix& other) {
        if(rows() != other.rows() || cols() != other.cols()) {
            throw std::invalid_argument(MATRICES_NOT_THE_SAME_SIZE);
        }

        const auto& otherInnerMat = other.innerMat();
        for(auto i = 0u; i < rows(); ++i){
            for(auto j = 0u; j < cols(); ++j) {
                mat[i][j] += otherInnerMat[i][j];
            }
        }
}



// ./Lab7/Matrix.h

#ifndef _MATRIX_H
```

```
#define _MATRIX_H

#include <vector>
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/vector.hpp>

class Matrix {
        public:
        Matrix()=default;
        Matrix(unsigned rows, unsigned cols);
        virtual ~Matrix()=default;

        public:
        using InnerMat = std::vector<std::vector<double>>;
        using InnerRow = std::vector<double>;

        public:
        virtual const InnerMat& innerMat() const;
        virtual InnerMat& innerMat();

        public:
        virtual unsigned rows() const;
        virtual unsigned cols() const;

        public:
        virtual void sum(const Matrix& other);

        public:
        friend std::ostream& operator<<(std::ostream& out, const Matrix& matrix);

        virtual InnerRow& operator[](unsigned index);
        virtual const InnerRow& operator[](unsigned index) const;
```

```
    private:
    friend class boost::serialization::access;
    template<class Archive>
        void serialize(Archive& ar, const unsigned int version) {
            ar & mat;
        }

    protected:
    InnerMat mat;
};


#endif //LAB6_MATRIX_H


// ./Lab7/MatrixFactory.cpp

#include "MatrixFactory.h"
#include "Matrix.h"

Matrix MatrixFactory::GenerateMatrix(unsigned rows, unsigned cols, double
minVal, double maxVal) {
    auto matrix = Matrix(rows, cols);
    for(auto i = 0u; i < matrix.rows();++i) {
        for(auto& el : matrix[i]) {
            el = (maxVal - minVal) * uni(rng) + minVal;
        }
    }
    return matrix;
}
```